

The Abstraction and Instantiation of String-Matching Programs *

Torben Amtoft

Computer Science Department, Boston University
111 Cummington Street, MCS, Boston, MA 02215, USA

Charles Consel

LaBRI / ENSEIRB

351, cours de la Libération, F-33405 Talence Cedex, France

Olivier Danvy

BRICS[†] Department of Computer Science, University of Aarhus
Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark

Karoline Malmkjær

Ericsson Telebit A/S

Skanderborgvej 232, DK-8260 Viby J, Denmark

April 2001 (this extended version: July 10, 2001)

Abstract

We consider a naive, quadratic string matcher testing whether a pattern occurs in a text; we equip it with a cache mediating its access to the text; and we abstract the traversal policy of the pattern, the cache, and the text. We then specialize this abstracted program with respect to a pattern, using the off-the-shelf partial evaluator Similix.

Instantiating the abstracted program with a left-to-right traversal policy yields the linear-time behavior of Knuth, Morris and Pratt's string matcher. Instantiating it with a right-to-left policy yields the linear-time behavior of Boyer and Moore's string matcher.

To Neil Jones, for his 60th birthday.

*Extended version of an article to appear in Neil Jones's Festschrift. Corresponding authors: Torben Amtoft (tamtft@cs.bu.edu) and Olivier Danvy (danvy@brics.dk).

[†]Basic Research in Computer Science (www.brics.dk), funded by the Danish National Research Foundation.

Contents

1	Background and introduction	4
1.1	Obtaining the KMP	4
1.2	This work	6
1.3	A note on program derivation and reverse engineering	7
1.4	Overview	8
2	The abstracted string matcher	8
2.1	The cache and its operations	8
2.2	The core program	11
3	Partial evaluation	14
3.1	Binding-time analysis	14
3.2	Polyvariant specialization	15
3.3	Post-unfolding	17
4	The KMP instances	17
5	The Boyer and Moore instances	18
5.1	The Boyer and Moore behavior	18
5.2	Pruning	19
5.2.1	No pruning:	19
5.2.2	Pruning once:	19
5.2.3	More pruning:	20
5.3	An example	20
6	Related work	20
6.1	Explicit management of static information	20
6.2	Implicit management of static information	22
6.3	Other derivations of Knuth, Morris and Pratt’s string matcher	22
6.4	Other derivations of Boyer and Moore’s string matcher	23
7	Conclusion and issues	23
A	Total correctness of the generic string matcher	25
A.1	Notation and properties	25
A.2	Invariants	26
A.3	Total correctness	27
A.4	Initialization	28
A.5	Return conditions	29
A.6	Preservation of invariants	29
A.7	Termination	32

List of Figures

1	The core quadratic string-matching program	12
2	Specialized version of Fig. 1 wrt. "aaa" à la Knuth, Morris and Pratt	18
3	Specialized version of Fig. 1 wrt. "abb" à la Boyer and Moore . .	21
4	Match-all counterpart of Fig. 3	24

1 Background and introduction

To build the first self-applicable partial evaluator [35, 36], Neil Jones, Peter Sestoft and Harald Søndergaard simplified its domain of discourse to an extreme: polyvariant specialization of symbolic first-order recursive equations. Source programs were expressed as recursive equations, data were represented as S-expressions, and the polyvariant-specialization strategy amounted to an interprocedural version of Kildal’s constant-propagation algorithm [3, 15]: for each of its calls, a recursive equation is specialized with respect to its known arguments and the resulting specialized equation is indexed with these known arguments and cached in a worklist. Specialization terminates once the worklist is complete, and the specialized equations in the worklist form the specialized program.

Since then, this basic framework has been subjected to many developments. For example, not all calls need to be specialized and give rise to a specialized recursive equation: many of them can merely be unfolded. Data need not be only symbolic either—numbers, characters, strings, pairs, higher-order functions, etc. can also be handled, and so can side effects. Finally, source programs need not be mere recursive equations—e.g., block-structured programs are amenable to partial evaluation as well.

Nevertheless, the basic strategy of polyvariant program-point specialization remains the same: generating mutually recursive residual equations, each of them corresponding to a source program point indexed by known values. In 1987, at the workshop on Partial Evaluation and Mixed Computation [9, 25], this very simple specialization strategy was challenged. Yoshihiko Futamura said that he had needed to design a stronger form of partial evaluation—Generalized Partial Computation [26]—to be able to specialize a naive quadratic-time string matcher into Knuth, Morris, and Pratt’s (KMP) linear-time string matcher [39]. He therefore asked the DIKU group whether polyvariant specialization was able to obtain the KMP.

1.1 Obtaining the KMP

The naive matching algorithm tests whether the pattern is the prefix of one of the suffixes of the text. Every time the pattern does not match a prefix of a suffix of the text, the pattern is shifted by one position to try the next suffix. In contrast, the KMP algorithm proceeds in two phases:

1. given the pattern, it constructs a ‘failure table’ in time linear in the size of the pattern; and
2. it traverses the text linearly, using the failure table to determine how much the pattern should be shifted in case of mismatch.

The first thing one could try is to specialize the traversal of the text with respect to a failure table. Unsurprisingly, the result is a linear-time residual program. In their article [39, page 330], Knuth, Morris and Pratt display a

similar program where the failure table has been “compiled” into the control flow.

More in the spirit of Futamura’s challenge, one can consider the naive specification, i.e., a (known) pattern string occurs in an (unknown) text if it is the prefix of one of its suffixes. On the left of each mismatch, the pattern and the text are equal:

```

      ---->
Pattern +-----o-----+
      ||||||||* mismatch
Text   +-----o-----+
      ---->

```

Yet, the naive specification shifts the pattern one place to the right and re-scans it together with the text up to the mismatch point and beyond:

```

      ---->   ---->
Pattern +-----o-----+
      |||||||||
Text   +-----o-----+
      ---->   ---->

```

Instead, one can match two instances of the pattern (the pattern and a prefix of the pattern equal to the prefix of the text matched so far, with one character chopped off) up to the point of mismatch and then resume scanning the pattern and the text:

```

      ---->
Pattern +-----o
      ||||||||---->
Pattern +-----o-----+
      ---->  |||||
Text     o-----+
      ---->

```

This staged program is equivalent to the original program. In particular, its complexity is the same. But if one specializes it with respect to a pattern, matching the pattern against itself is carried out at partial-evaluation time. Furthermore, since the index on the text only increases, specializing this staged program yields a string matcher that does not back up on the text and thus operates in time linear in the text. Therefore part of the challenge is met: polyvariant specialization can turn a (staged) quadratic string matcher into a linear one.

But how does this linear string matcher compare to the KMP?

Short of a formal proof, one can compare, for given patterns, (1) the corresponding residual string matchers and (2) the results of specializing the second phase of the KMP with respect to the corresponding failure tables produced by the first phase. It turns out that the resulting string matchers are indeed isomorphic, suggesting that mere polyvariant specialization can obtain the KMP if the naive string matcher is staged as pictured above [18].

This experiment clarifies that linearity requires one to keep a static view of the dynamic prefix of the text during specialization. This static view can be kept explicitly in the source program, as above. Alternatively, a partial evaluator could implicitly memoize the outcome of every dynamic test. (For example, given a dynamic variable x , a conditional expression `(if (odd? x) e1 e2)`, and some knowledge about `odd?`, such a partial evaluator would know that x is odd when processing `e1` and that x is even when processing `e2`.) Futamura’s Generalized Partial Computation is such a partial evaluator. Therefore, it automatically keeps a static view of the dynamic prefix of the text during specialization and thus obtains the KMP out of a naive, unstaged string matcher. So the key to linearity is precisely the static view of the dynamic prefix of the text during specialization.

To summarize, we need a static view of the dynamic prefix of the text during specialization. Whether this view is managed implicitly in the partial evaluator (as in Generalized Partial Computation) or explicitly in the source program (as in basic polyvariant specialization) is a matter of convenience. If the management is implicit, the program is simpler but the partial evaluator is more complex. If the management is explicit, the program is more complicated but the partial evaluator is simpler.

That said, there is more to the KMP than linearity over the text—there is also linearity over the pattern. Indeed, the KMP algorithm operates on a failure table that is constructed in time linear in the size of the pattern, whereas in general, a partial evaluator does not operate in linear time.

1.2 This work

We extend the construction of Section 1.1 to obtain Knuth, Morris, and Pratt’s linear string matcher as well as string matchers in the style of Boyer and Moore from the same quadratic string matcher.

To this end, we instrument the naive string matcher with a *cache* that keeps a trace of what is known about the text, similarly to a memory cache on a hardware processor. Any access to the text is mediated by an access to the cache. If the information is already present in the cache, the text is not accessed. If the information is not present in the cache, the text is accessed and the cache is updated.

The cache has the same length as the pattern and it gives us a static view of the text. We make it keep track both of what is known to occur in the text and of what is known not to occur in the text. Each entry of the cache thus contains either some *positive* information, namely the corresponding character in the text, or some *negative* information, namely a list of (known) characters that do not match the corresponding (unknown) character in the text. The positive information makes it possible to test characters statically, i.e., without accessing the text. The negative information makes it possible to detect statically when a test would fail. Initially, the cache contains no information (i.e., each entry contains an empty list of negative information).

By construction, the cache-based string matcher consults an entry in the text either not at all or repeatedly until the entry is recognized. An entry in the text can be left unconsulted because an adjacent character does not match the pattern. Once an entry is recognized, it is never consulted again because the corresponding character is stored in the cache. The entry can be consulted repeatedly, either until it is recognized or until the pattern has shifted. Each unrecognized character is stored in the cache and thus the number of times an entry is accessed is at most the number of different characters in the pattern. Therefore, each entry in the text is consulted a bounded number of times. The specialized versions of the cache-based string matcher inherit this property, and therefore their matching time is linear with respect to the length of the text.

On the other hand, the size of the specialized versions can be large: poly-variant specialization yields residual equations corresponding to source program points indexed by known values—here, specifically, the cache. In order to reduce the size of the residual programs, we allow ourselves to *prune* the cache in the source string matcher. For example, two extreme choices are to remove all information and to remove no information from the cache, but variants are possible.

Finally, since the KMP compares the pattern and the text from left to right whereas the Boyer and Moore compares the pattern and the text from right to left, we also parameterize the string matcher with a traversal policy. For example, two obvious choices are left to right and right to left, but variants are possible.

Our results are that traversing the pattern from left to right yields the KMP behavior, with variants, and that traversing the pattern from right to left yields the Boyer and Moore behavior [13, 14], also with variants. (The variants are determined by how we traverse the pattern and the text and how we prune the cache.) These results are remarkable (1) because they show that it is possible to obtain both the KMP linear behavior and the Boyer and Moore linear behavior from the *same* source program, and (2) because even though it is equipped with a cache, this source program is a naive quadratic string-matching program.

The first author is aware of these results since the early 1990s [5] and the three other authors since the late 1980s [20]. Since then, these results have been reproduced and extended by Queinnec and Geffroy [46], but otherwise they have only been mentioned informally [19, Section 6.1].

1.3 A note on program derivation and reverse engineering

We would like to point out that we are not using partial evaluation to reverse-engineer the KMP and the Boyer and Moore string matchers. What we are attempting to do here is to exploit the simple observation that, in the naive quadratic string matcher, and as depicted in Section 1.1, rematching can be optimized away by partial evaluation.

In 1988, we found that the resulting specialized programs behave like the KMP, for a left-to-right traversal. We then conjectured that for a right-to-left traversal, the resulting specialized programs should behave like Boyer and

Moore—which they do. In fact, for any traversal, partial evaluation yields a linear-time residual program (which may be sizeable), at least as long as no pruning is done. We have observed that the method scales up to patterns with wild cards and variables as well as to pattern matching in trees à la Hoffman and O’Donnell [32]. For two other examples, the method has led Queinnec and Geffroy to derive Aho and Corasick’s as well as Comments and Walter’s string matchers [46].

1.4 Overview

The rest of this article is organized as follows. Section 2 presents the core program and its parameters. Section 3 specifies the action of partial evaluation on the core program. Section 4 describes its KMP instances and Section 5 describes its Boyer and Moore instances. Related work is reviewed in Section 6 and Section 7 concludes. A correctness proof of the core program can be found in Appendix A.

Throughout, we use the programming language Scheme [37], or, more precisely, Petite Chez Scheme (www.scheme.com) and the partial evaluator Simlix [11, 12].

2 The abstracted string matcher

We first specify the operations on the cache (Section 2.1). Then we describe the cache-based string matcher (Section 2.2).

2.1 The cache and its operations

The cache has the same size as the pattern. It holds a picture of what is currently known about the text: either we know a character, or we know nothing about a character, or we know that a character is not equal to some given characters. Initially, we know nothing about any entry of the text. In the course of the algorithm, we may get to know some characters that do not match an entry, and eventually we may know this entry. We then disregard the characters that do not match the entry, and we only consider the character that matches the entry. Accordingly, each entry of the cache contains either some positive information (one character) or some negative information (a list of characters, possibly empty). We test this information with the following predicates.

```
(define pos? char?)

(define (neg? e)
  (or (null? e) (pair? e)))
```

These two predicates are mutually exclusive.

We implement the cache as a list and we operate on it as follows.

- Given a natural number specifying the length of the pattern, `cache-init` constructs an initial cache containing empty lists of characters.

```
(define (cache-init n)
  (if (= n 0)
      '()
      (cons '() (cache-init (- n 1)))))
```

- In the course of string matching, the pattern slides to the right of the text by one character. Paralleling this shift, `cache-shift` maps a cache to the corresponding shifted cache where the right-most entry is negative and empty.

```
(define (cache-shift c)
  (append (cdr c) (list '())))
```

- Given an index, the predicates `cache-ref-pos?` and `cache-ref-neg?` test whether a cache entry is positive or negative.

```
(define (cache-ref-pos? c i)
  (pos? (list-ref c i)))

(define (cache-ref-neg? c i)
  (neg? (list-ref c i)))
```

- Positive information is fetched by `cache-ref-pos` and negative information by `cache-ref-neg`.

```
(define cache-ref-pos list-ref)

(define cache-ref-neg list-ref)
```

- Given a piece of positive information at an index, `cache-extend-pos` extends the cache with this positive information at that index. The result is a new cache where the positive information supersedes any negative information at that index.

```
(define (cache-extend-pos c i pos)
  (letrec ([walk
            (lambda (c i)
              (if (= i 0)
                  (cons pos (cdr c))
                  (cons (car c) (walk (cdr c) (- i 1))))))]
    (walk c i)))
```

- Given more negative information at an index, `cache-extend-neg` extends the cache with this negative information at that index. The result is a new cache.

```

(define (cache-extend-neg c i neg)
  (letrec ([walk
            (lambda (c i)
              (if (= i 0)
                  (cons (cons neg (car c)) (cdr c))
                  (cons (car c) (walk (cdr c) (- i 1))))))]
    (walk c i)))

```

- Given the cache, `schedule-pc` and `schedule-pt` respectively yield the series of indices (represented as a list) through which to traverse the pattern and the cache, and through which to traverse the pattern and the text, respectively. The formal requirements on `schedule-pc` and `schedule-pt` are stated in Appendix A.

1. The following definition of `schedule-pc` specifies a left-to-right traversal of the pattern and the cache:

```

(define (schedule-pc c)
  (letrec ([walk
            (lambda (i c)
              (cond
                [(null? c)
                 '()]
                [(pos? (car c))
                 (cons i (walk (+ i 1) (cdr c)))]
                [else
                 (if (null? (car c))
                     (walk (+ i 1) (cdr c))
                     (cons i (walk (+ i 1) (cdr c))))])]
            (walk 0 c)))

```

For example, applying `schedule-pc` to an empty cache yields the empty list. This list indicates that there is no need to compare the pattern and the cache.

For another example, applying `schedule-pc` to a cache containing two pieces of positive information, and then one non-empty piece of negative information, and then three empty pieces of negative information yields the list `(0 1 2)`. This list indicates that the pattern and the cache should be successively compared at indices 0, 1, and 2, and that the rest of the cache should be ignored.

2. The following definition of `schedule-pt` specifies a left-to-right traversal of the pattern and of the text:

```

(define (schedule-pt c)
  (letrec ([walk
            (lambda (i c)
              (cond
                [(null? c)
                 '()]
                [(pos? (car c))
                 (walk (+ i 1) (cdr c))]
                [else
                 (cons i (walk (+ i 1) (cdr c)))]))]
    (walk 0 c)))

```

For example, applying `schedule-pt` to an empty cache of size 3 (as could be obtained by evaluating `(cache-init 3)`) yields the list `(0 1 2)`. This list indicates that the pattern and the text should be successively compared at indices 0, 1, and 2.

For another example, applying `schedule-pt` to a cache containing two pieces of positive information, then one non-empty piece of negative information, and then three empty pieces of negative information yields the list `(2 3 4 5)`. This list indicates that the pattern and the text already agree at indices 0 and 1, and should be successively compared at indices 2, 3, 4, and 5. (The negative information at index 2 in the cache is of no use here.)

- Finally, we give ourselves the possibility of pruning the cache with `cache-prune`. Pruning the cache amounts to shortening lists of negative information and resetting positive information to the empty list of negative information. For example, the identity function prunes nothing at all:

```

(define (cache-prune c)
  c)

```

2.2 The core program

The core program, displayed in Figure 1, is a cache-based version of a naive quadratic program checking whether the pattern occurs in the text, i.e., if the pattern is a prefix of one of the successive suffixes of the text, from left to right. The program is composed of two mutually recursive loops: a static one checking whether the pattern matches the cache and a dynamic one checking whether the pattern matches the rest of the text.

The main function, `match`: Initially, `match` is given a pattern `p` and a text `t`. It computes their length (`1p` and `1t`, respectively), and after checking that there is room in the text for the pattern, it initiates the dynamic loop with an empty cache. The result is either `-1` if the pattern does not occur in the text, or the index of the left-most occurrence of the pattern in the text.

```

(define (match p t)
  (let ([lp (string-length p)])      ;; lp is the length of p
    (if (= lp 0)
        0
        (let ([lt (string-length t)] ;; lt is the length of t
              (if (< lt lp)
                  -1
                  (match-pt p lp (cache-init lp) t 0 lt)))))))

(define (match-pc p lp c t bt lt)
  ;; matches p[0..lp-1] and c[0..lp-1]
  (letrec ([loop-pc
            (lambda (z c is)
              (if (null? is)
                  (let ([bt (+ bt z)] [lt (- lt z)])
                    (if (< lt lp)
                        -1
                        (match-pt p lp c t bt lt)))
                  (let ([i (car is)])
                    (if (if (cache-ref-pos? c i)
                            (equal? (string-ref p i)
                                     (cache-ref-pos c i))
                            (not (member (string-ref p i)
                                         (cache-ref-neg c i))))
                        (loop-pc z c (cdr is))
                        (let ([c (cache-shift c)])
                          (loop-pc (+ z 1) c (schedule-pc c))))))))])
    (loop-pc 1 c (schedule-pc c))))

(define (match-pt p lp c t bt lt)
  ;; matches p[0..lp-1] and t[bt..bt+lp-1]
  (letrec ([loop-pt
            (lambda (c is)
              (if (null? is)
                  bt
                  (let* ([i (car is)] [x (string-ref p i)])
                    (if (equal? (string-ref t (+ bt i)) x)
                        (loop-pt (cache-prune
                                 (cache-extend-pos c i x))
                                (cdr is))
                        (match-pc p lp
                                 (cache-shift
                                  (cache-extend-neg c i x))
                                 t bt lt))))))]
    (loop-pt c (schedule-pt c))))

```

Figure 1: The core quadratic string-matching program

The static loop, `match-pc`: The pattern is iteratively matched against the cache using `loop-pc`. For each mismatch, the cache is shifted and `loop-pc` is resumed. Eventually, the pattern agrees with (i.e., matches) the cache—if only because after repeated iterations of `loop-pc` and shifts of the cache, the cache has become empty. Then, if there is still space in the text for the pattern, the dynamic loop is resumed.

In more detail, `match-pc` is passed the pattern `p`, its length `lp`, the cache `c`, and also the text `t`, its base index `bt`, and the length `lt` of the remaining text to match in the dynamic loop. The static loop checks iteratively (with `loop-pc`) whether the pattern and the cache agree. The traversal takes place in the order specified by `schedule-pc` (e.g., left to right or right to left). For each index, `loop-pc` tests whether the information in the cache is positive or negative. In both cases, if the corresponding character in the pattern agrees (i.e., either matches or does not mismatch), `loop-pc` iterates with the next index. Otherwise the cache is shifted and `loop-pc` iterates. In addition, `loop-pc` records in `z` how many times the cache has been shifted (initially 1 since `match-pc` is only invoked from the dynamic loop). Eventually, the pattern and the cache agree. The new base and the new length of the text are adjusted with `z`, and if there is enough room in the text for the pattern to occur, the dynamic loop takes over.

When the dynamic loop takes over, the pattern and the cache completely agree, i.e., for all indices i :

- either the cache information is positive at index i and it contains the same character as the pattern at index i ;
- or the cache information is negative at index i and the character at index i in the pattern does not occur in the list of characters contained in this negative information.

The dynamic loop, `match-pt`: The pattern is iteratively matched against the parts of the text that are not already in the cache, using `loop-pt`. At each iteration, the cache is updated. If a character matches, the cache is updated with the corresponding positive information before the next iteration of `loop-pt`. If a character does not match, the cache is updated with the corresponding negative information and the static loop (i.e., `match-pc`) is resumed with a shifted cache.

In more detail, `match-pt` is passed the pattern `p`, its length `lp`, the cache `c`, and the text `t`, its base index `bt`, and the length `lt` of the remaining text to match. The traversal takes place in the order specified by `schedule-pt` (e.g., left to right or right to left). For each index, `loop-pt` tests the corresponding character in the text against the corresponding character in the pattern. If the two characters are equal, then `loop-pt` iterates with an updated and pruned cache. Otherwise, the static loop takes over with an updated and shifted cache. If `loop-pt` runs out of indices, pattern matching succeeds and the result is the base index in the text.

The cache: The cache is only updated (positively or negatively) during the dynamic loop. Furthermore, it is only updated with a character *from the pattern*

and never with one from the text. Considering that the pattern is known (i.e., static) and that the text is unknown (i.e., dynamic), constructing the cache with characters from the pattern ensures that it remains known at partial-evaluation time. This selective update of the cache is the key for successful partial evaluation.

We allow `schedule-pt` to return indices of characters that are already in the cache and to return a list of indices with repetitions. These options do not invalidate the correctness proof of the core program, but they make it possible to reduce the size of residual programs, at the expense of redundant tests. This information-theoretic weakening is common in published studies of the Boyer-Moore string matcher [6, 33, 45]: the challenge then is to show that the weakened string matcher still operates in linear time [47].

In the current version of Figure 1, we only prune the cache at one point: after a call to `cache-extend-pos`. This lets us obtain a number of variants of string matchers in the style of Boyer and Moore. But as noted in Section 5.2, at least one remains out of reach.

3 Partial evaluation

Specializing the core string matcher with respect to a pattern takes place in the traditional three steps: preprocessing (binding-time analysis), processing (specialization), and postprocessing (unfolding).

3.1 Binding-time analysis

Initially, `p` is static and `t` is dynamic.

In `match`, `lp` is static since `p` is static. The first `let` expression and the first conditional expression are thus static. Conversely, `lt` is dynamic since `t` is dynamic. The second `let` expression and the second conditional expression are thus dynamic. After binding-time analysis, `match` is annotated as follows.

```
(define (match ps td)
  (lets ([lps (string-length ps)])
    (ifs (= lps 0s)
      0d
      (letd ([ltd (string-length td)])
        (ifd (< ltd lps)
          -1d
          (match-pt ps lps (cache-init lps) td 0d ltd))))))
```

In `match-pc` and in `match-pt`, `p`, `lp` and `c` are static, and `t`, `bt`, and `lt` are dynamic. (Therefore, `0` is generalized to be dynamic in the initial call to `match-pt`. Similarly, since `match` returns a dynamic result, `0` and `-1` are also generalized to be dynamic.)

In `loop-pc`, `c` and `is` are static and—due to the automatic poor man’s generalization of Similix [34]—`z` is dynamic. Except for the conditional expression

testing (`< lt lp`) and its enclosing `let` expression, which are dynamic, all syntactic constructs are static.

After binding-time analysis, `match-pc` and `loop-pc` are annotated as follows.

```
(define (match-pc ps lps cs td btd ltd)
  (letrec ([loop-pc
            (lambda (zd cs iss)
              (ifs (null? iss)
                    (letd ([btd (+ btd zd)] [ltd (- ltd zd)])
                      (ifd (< ltd lps)
                            -1d
                            (match-pt ps lps cs td btd ltd)))
                    (lets ([is (car iss)]
                          (ifs (ifs (cache-ref-pos? cs is)
                                  (equal? (string-ref ps is)
                                           (cache-ref-pos cs is))
                                  (not (member (string-ref ps is)
                                             (cache-ref-neg cs is))))
                            (loop-pc zd cs (cdr iss))
                            (lets ([cs (cache-shift cs)]
                                  (loop-pc (+ zd 1d) cs (schedule-pc cs))))))
              (loop-pc 1d cs (schedule-pc cs))))))
```

In `loop-pt`, `c` and `is` are static. Except for the conditional expression performing an equality test, all syntactic constructs are static.

After binding-time analysis, `match-pt` and `loop-pt` are annotated as follows.

```
(define (match-pt ps lps cs td btd ltd)
  (letrec ([loop-pt
            (lambda (cs iss)
              (ifs (null? iss)
                    btd
                    (let*s ([is (car iss)] [xs (string-ref ps is)]
                          (ifd (equal? (string-ref td (+ btd is)) xs)
                                (loop-pt (cache-prune
                                          (cache-extend-pos cs is xs)
                                          (cdr iss))
                                          (match-pc ps lps
                                                    (cache-shift
                                                      (cache-extend-neg cs is xs)
                                                      td btd ltd))))))
              (loop-pt cs (schedule-pt cs))))))
```

3.2 Polyvariant specialization

In Similix [12], the default specialization strategy is to unfold all function calls and to treat every conditional expression with a dynamic test as a specialization point. Here, since there is exactly one specialization point in `loop-pc` as well as in `loop-pt`, without loss of generality, we refer to each instance of a specialization point as *an instance of loop-pc* and *an instance of loop-pt*, respectively.

Each instance of `loop-pc` is given a base index in the text, the remaining length of the text, and a natural number. The body of this instance is a clone of the dynamic `let` expression and of the dynamic conditional expression in the original `loop-pc`: it increments `bt` and decrements `lt` with the natural number, checks whether the remaining text is large enough, and calls an instance of `loop-pt`, as specified by the following template (where `<lp>` is a natural number denoting the length of the pattern).

```
(define (loop-pc-... t bt lt z)
  (let ([bt (+ bt z)] [lt (- lt z)])
    (if (< lt <lp>)
        -1
        (loop-pt-... t bt lt))))
```

Each instance of `loop-pt` is given a base index in the text and the remaining length of the text (their sum is always equal to the length of the text). It tests whether a character in the text is equal to a fixed character. If so, it either returns an index or branches to another instance of `loop-pt`. If not, it branches to an instance of `loop-pc` with a fixed shift. (Below, `<i>` stands for a non-negative integer, `<n>` stands for a natural number, and `<c>` stands for a character.)

```
(define (loop-pt-... t bt lt)
  (if (equal? (string-ref t (+ bt <i>)) <c>)
      bt
      (loop-pc-... t bt lt <n>)))

(define (loop-pt-... t bt lt)
  (if (equal? (string-ref t (+ bt <i>)) <c>)
      (loop-pt-... t bt lt)
      (loop-pc-... t bt lt <n>)))
```

Therefore, a residual program consists of mutually recursive specialized versions of the two specialization points, and of a main function computing the length of the text, checking whether it is large enough, and calling a specialized version of `loop-pt`. The body of each auxiliary function is constructed out of instances of the dynamic parts of the programs, i.e., it fits one of the templates above [21, 42].

We have written the holes in the templates between brackets. The name of each residual function results from concatenating `loop-pc-` and `loop-pt-` to a fresh index. Each residual function is closed, i.e., it has no free variables.

Overall, the shape of a residual program is as follows.

```
(define (match-0 t)
  (let ([lt (string-length t)])
    (if (< lt ...)
        -1
        (loop-pt-1 t 0 lt))))
```

```

(define (loop-pc-1 t bt lt z) ...)

(define (loop-pc-2 t bt lt z) ...)

...

(define (loop-pt-1 t bt lt) ...)

(define (loop-pt-2 t bt lt) ...)

...

```

This general shape should make it clear how residual programs relate to (and how their control flow could be “decompiled” into) a KMP-like failure table, as could be obtained by data specialization [7, 16, 38, 41].

3.3 Post-unfolding

To make the residual programs more readable, Similix post-unfolds residual functions that are called only once. It also defines the remaining functions locally to the main residual function. (By the same token, it could lambda-drop the variable `t`, as we do in the residual programs displayed in Sections 4, 5, and 7, for readability [23].)

So all in all, a specialized program is defined as a main function (an instance of `match`) with many locally defined and mutually recursive auxiliary functions (instances of `loop-pc` and `loop-pt`), each of which is called more than once.

4 The KMP instances

In the KMP style of string matching, the pattern and the corresponding prefix of the text are traversed from left to right. Therefore, we define `schedule-pt` so that `match-pt` proceeds from left to right. The resulting program is still quadratic, but specializing it with respect to a pattern yields a residual program traversing the text in linear time if the cache is not pruned. Furthermore, all residual programs are independent of `schedule-pc` since it is applied at partial-evaluation time. Since Consel and Danvy’s work [18], this traversal has been consistently observed to coincide with the traversal of Knuth, Morris and Pratt’s string matcher. (We are not aware of any formal proof of this coincidence, but we expect that we will be able to show that the two matchers operate in lock step.)

This ‘KMP behavior’ has been already described in the literature (see Section 6), so we keep this section brief. In this instantiation, the cache is naturally composed of three parts: a left part with positive information, a right part with empty negative information, and between the left part and the right part, at most one entry with non-empty negative information. This entry may or may not be pruned away, which affects both the size of the residual code and its runtime, as analyzed by Grobauer and Lawall [30].

```

(define (match-aaa t)
  (define (loop-pc-1 bt lt z)
    (let ([bt (+ bt z)] [lt (- lt z)])
      (if (< lt 3)
          -1
          (loop-pt-1 bt lt))))
  (define (loop-pt-1 bt lt)
    (if (equal? (string-ref t (+ bt 0)) #\a)
        (if (equal? (string-ref t (+ bt 1)) #\a)
            (if (equal? (string-ref t (+ bt 2)) #\a)
                bt
                (loop-pc-1 bt lt 3))
            (loop-pc-1 bt lt 2))
        (loop-pc-1 bt lt 1)))
  (let ([lt (string-length t)])
    (if (< lt 3)
        -1
        (loop-pt-1 0 lt))))

```

- For all t , `(match-aaa t)` equals `(match "aaa" t)`.
- For all z , `(loop-pc-1 bt lt z)` equals `(loop-pc z '(() () ()) '())` evaluated in the scope of `bt` and `lt`.
- `(loop-pt-1 bt lt)` equals `(loop-pt '(() () ()) '(0 1 2))` evaluated in the scope of `bt` and `lt`.

Figure 2: Specialized version of Fig. 1 wrt. "aaa" à la Knuth, Morris and Pratt

Figure 2 displays a specialized version of the core program in Figure 1 with respect to the pattern string "aaa". This specialized version is renamed for readability. We instantiated `cache-prune` to the identity function. The code for `loop-pt-1` reveals that a left-to-right strategy is indeed employed, in that the text is addressed with offsets 0, 1, and 2.

5 The Boyer and Moore instances

5.1 The Boyer and Moore behavior

In the Boyer and Moore style of string matching, the pattern and the corresponding prefix of the text are traversed from right to left. Therefore, we define `schedule-pt` so that `match-pt` proceeds from right to left. The resulting program is still quadratic, but specializing it with respect to a pattern yields a residual program traversing the text in linear time if the cache is not pruned and again independently of `schedule-pc`, which is executed at partial-evaluation time. We

have consistently observed that this traversal coincides with the traversal of string matchers à la Boyer and Moore,¹ provided that `schedule-pt` processes the non-empty negative entries in the cache before the empty ones. Then there will always be at most one entry with non-empty negative information.

Except for the first author’s PhD dissertation [5], this “Boyer and Moore behavior” has not been described in the literature, so we describe it in some detail here. Initially, in Figure 1, `loop-pt` iteratively builds a suffix of positive information. This suffix is located on the right of the cache, and grows from right to left. In case of mismatch, `loop-pt` punctuates the suffix with one (non-empty) negative entry, shifts the cache to the right, and resumes `match-pc`. The shift transforms the suffix into a segment, which drifts to the left of the cache while `loop-pt` and `loop-pc` continue to interact. Subsequently, if a character match is successful, the negative entry in the segment becomes positive, and the matcher starts building up a new suffix. The cache is thus composed of zero or more segments of positive information, the right-most of which may be punctuated on the left by one negative entry. Each segment is the result of an earlier unsuccessful call to `loop-pt`. The right-most segment (resp. the suffix), is the witness of the latest (resp. the current) call to `loop-pt`. In the course of string matching, adjacent segments can merge into one.

5.2 Pruning

5.2.1 No pruning:

Never pruning the cache appears to yield Knuth’s optimal Boyer and Moore string matcher [39, page 346].

5.2.2 Pruning once:

Originally [13], Boyer and Moore maintained two tables. The first table indexes each character in the alphabet with the right-most occurrence (if any) of this character in p . The second table indexes each position in p with the rightmost occurrence (if any) of the corresponding suffix, preceded by a different character, elsewhere in p .

We have not obtained Boyer and Moore’s original string matcher because it exploits the two tables in a rather unsystematic way. Nevertheless, the following instantiation appears to yield a simplified version of Boyer and Moore’s original string matcher, where only the first table is used (and hence where linearity does not hold):

- `cache-prune` empties the cache;
- after listing the entries with non-empty negative information, `schedule-pt` lists *all* negative entries, even those (zero or one) that are already listed.

¹Again, we expect to be able to prove this coincidence by showing that the two traversals proceed in lock step.

(This redundancy is needed in order to keep the number of residual functions small, at the expense of redundant dynamic (residual) tests.)

Partsch and Stomp also observed that Boyer and Moore’s original string matcher uses the two tables unsystematically [45]. A more systematic take led them to formally derive the alternative string matcher hinted at by Boyer and Moore in their original article [13, page 771]. It appears that this string matcher can be obtained as follows:

- `cache-prune` removes all information except for the right-most suffix of positive information (i.e., the maximal set of the form $\{j, \dots, lp - 1\}$ with all the corresponding entries being positive);
- after having listed the non-empty negative information, `schedule-pt` lists *all* indices (including the positive ones).

5.2.3 More pruning:

The pruning strategy of Figure 1 is actually sub-optimal. A better one is to prune the cache before reaching each specialization point. Doing so makes it possible to obtain what looks like Horspool’s variant of Boyer and Moore’s string matcher [33].

5.3 An example

Figure 3 displays a specialized version of the program in Figure 1 with respect to the pattern string "abb". This specialized version is renamed for readability. We used instantiations yielding Partsch and Stomp’s variant. The code for `loop-pt-1` reveals that a right-to-left strategy is indeed employed, in that the text is addressed with offsets 2, 1, and 0. (NB. Similix has pretty-printed successive `if` expressions into one `cond` expression.)

6 Related work

In his MS thesis [49, Sec. 8.2], Sørensen has observed that once the pattern is fixed, the naive string matcher is de facto linear—just with a factor proportional to the length of this pattern, lp . The issue of obtaining the KMP behavior is therefore to produce a specialized program that runs in linear time with a factor independent of lp . Since Consel and Danvy’s original solution [18], obtaining the KMP behavior by partial evaluation (i.e., specialized programs that do not backtrack on the text) has essentially followed two channels: managing static information explicitly vs. managing static information implicitly.

6.1 Explicit management of static information

A number of variants of the naive string matcher have been published that keep a static trace of the dynamic prefix explicitly [5, 34, 50, 51]. However, as the

```

(define (match-abb t)
  (define (loop-pc-1 bt lt z)
    (let ([bt (+ bt z)] [lt (- lt z)])
      (if (< lt 3)
          -1
          (loop-pt-1 bt lt))))
  (define (loop-pt-1 bt lt)
    (if (equal? (string-ref t (+ bt 2)) #\b)
        (if (equal? (string-ref t (+ bt 1)) #\b)
            (if (equal? (string-ref t (+ bt 0)) #\a)
                bt
                (loop-pc-1 bt lt 3))
            (let ([bt (+ bt 1)] [lt (- lt 1)])
              (cond
               [(< lt 3)
                -1]
               [(equal? (string-ref t (+ bt 0)) #\a)
                (loop-pt-1 bt lt)]
               [else
                (loop-pc-1 bt lt 2)]))))
        (let ([bt (+ bt 2)] [lt (- lt 2)])
          (cond
           [(< lt 3)
            -1]
           [(equal? (string-ref t (+ bt 0)) #\a)
            (loop-pt-1 bt lt)]
           [else
            (loop-pc-1 bt lt 1)]))))))
  (let ([lt (string-length t)])
    (if (< lt 3)
        -1
        (loop-pt-1 0 lt))))

```

- For all t , $(\text{match-abb } t)$ equals $(\text{match "abb" } t)$.
- For all z , $(\text{loop-pc-1 } bt \ lt \ z)$ equals $(\text{loop-pc } z \ '(() \ () \ ()) \ '())$ evaluated in the scope of bt and lt ,
- $(\text{loop-pt-1 } bt \ lt)$ equals $(\text{loop-pt } '(() \ () \ ()) \ '(2 \ 1 \ 0))$ evaluated in the scope of bt and lt .

Figure 3: Specialized version of Fig. 1 wrt. "abb" à la Boyer and Moore

first author put it in his PhD thesis, “it is not obvious that [the transformation] preserves semantics” [5, page 176]. Indeed, experience has shown that the transformation is error-prone. Therefore, in Appendix A, we have spelled out the correctness proof of the generic string matcher. Also, recently, Grobauer and Lawall have revisited how to obtain the KMP behavior by explicitly managing the static prefix, and have proven its correctness [30].

Another concern is the size of specialized programs. As Consel and Danvy put it, “there are no guarantees about the size of these programs, nor about the time taken by the partial evaluator to produce them” [19]. But Grobauer and Lawall have shown that specialized programs with the KMP behavior have a size linear in the length of the pattern [30]. As for the resources (time and space) taken by the partial evaluator, polyvariant specialization does not work in linear time in general. Therefore, it does not produce the specialized program in linear time, in contrast to Knuth, Morris and Pratt’s algorithm, which first constructs a failure table in time linear to the pattern string.

6.2 Implicit management of static information

A number of partial evaluators have been developed that keep a static trace of the dynamic prefix implicitly, making them able to pass the “KMP test” [49], i.e., to specialize the original quadratic string-matching program into a KMP-like residual program. Such partial evaluators include Futamura’s Generalized Partial Computation [26], Smith’s partial evaluator for constraint logic programming languages [48], Queindec and Geffroy’s intelligent backtracking [46], supercompilation [28, 29, 49, 50, 51], partial deduction [44], partial evaluators for functional logic programs [4, 40], and the composition of a memoizing interpreter and a standard partial evaluator [27].

Like Similix, none of these partial evaluators has been proven correct and there are no guarantees about the resources required to produce residual programs and the size of the residual programs. Nevertheless, all of them have been tested to produce an output similar to the output of a simple partial evaluator over a naive string matcher that keeps a static trace of the dynamic prefix explicitly.

6.3 Other derivations of Knuth, Morris and Pratt’s string matcher

Reconstructing the KMP appears to be a favorite in the program-transformation community, in some sense following Knuth’s steps since he obtained the behavior of the KMP by calculating it from Cook’s construction [39, page 338]. Examples include Dijkstra’s use of invariants [24], Bird’s tabulation technique [8], Takeichi and Akama’s equational reasoning [53], Colussi’s use of Hoare logic [17], and just recently Hernández and Rosenblueth’s logic-program derivation [31]. Further variants of the KMP can be found, e.g., in Watson’s PhD thesis [54] and in Crochemore and Hancart’s chapter in the *Handbook of Algorithms and Theory of Computation* [22].

6.4 Other derivations of Boyer and Moore’s string matcher

We have only found two reconstructions of Boyer and Moore’s string matcher in the literature: Partsch and Stomp’s formal derivation [45] and just recently Hernández and Rosenblueth’s logic-program derivation [31]. But as reviewed in Aho’s chapter in the *Handbook of Theoretical Computer Science* [1], several variants of Boyer and Moore’s string matcher exist (such as Sunday’s variant [52] and Baeza-Yates, Choffrut, and Gonnet’s variant [6]), with recurrent concerns about linearity in principle (e.g., Schaback’s work [47]) and in practice (e.g., Horspool’s work [33]).

7 Conclusion and issues

We have abstracted a naive quadratic substring program with a static cache, and illustrated how specializing various instances of it with respect to a pattern gives rise to KMP-like and Boyer-Moore-like linear-time string matchers. This use of partial evaluation amounts to preprocessing the pattern by program specialization. This preprocessing is not geared to be efficient, since we use an off-the-shelf partial evaluator, but we find it illuminating as a guide for exploring pattern matching in strings.

Generalizing, one can extend the core string matcher to yield the list of all the matches instead of yielding (the index of) the left-most match. One can easily show that the sizes of the specialized programs do not change, as witnessed by Figure 4 that displays the match-all counterpart of the match-leftmost Figure 3.

Shifting perspective, one can also consider specializing the naive quadratic substring program with respect to a text instead of with respect to a pattern. One can then equip this program with a static cache, *mutatis mutandis*, and specialize various instances of it with respect to a text, obtaining programs that traverse the pattern in linear time. In the late 1980s [41, 43], Malmkjær and Danvy observed that traversing the pattern and the text from left to right yields a program representation of suffix trees [55], i.e., position trees [2], and that suitably pruning the cache yields the smallest automaton recognizing the subwords of a text [10]. Traversing the pattern and the text from right to left, however, is not possible, since we are not given the pattern and thus we do not know its length. Yet we can use the trick of enumerating its possible lengths, or again, more practically, we can be given its length as an extra static information. Partial evaluation then gives rise to a program representation of trees that are to Boyer and Moore what position trees are to Knuth, Morris and Pratt.

```

(define (match-all-abb t)
  (define (loop-pc-1 bt lt z)
    (let ([bt (+ bt z)] [lt (- lt z)])
      (if (< lt 3)
          '()
          (loop-pt-1 bt lt))))
  (define (loop-pt-1 bt lt)
    (if (equal? (string-ref t (+ bt 2)) #\b)
        (if (equal? (string-ref t (+ bt 1)) #\b)
            (if (equal? (string-ref t (+ bt 0)) #\a)
                (cons bt (loop-pc-1 bt lt 3)) ; collection
                (loop-pc-1 bt lt 3))
            (let ([bt (+ bt 1)] [lt (- lt 1)])
              (cond
               [(< lt 3)
                '()]
               [(equal? (string-ref t (+ bt 0)) #\a)
                (loop-pt-1 bt lt)]
               [else
                (loop-pc-1 bt lt 2)])))
        (let ([bt (+ bt 2)] [lt (- lt 2)])
          (cond
           [(< lt 3)
            '()]
           [(equal? (string-ref t (+ bt 0)) #\a)
            (loop-pt-1 bt lt)]
           [else
            (loop-pc-1 bt lt 1)]))))
  (let ([lt (string-length t)])
    (if (< lt 3)
        '()
        (loop-pt-1 0 lt))))

```

Figure 4: Match-all counterpart of Fig. 3

Getting back to the original motivation of this article, we would like to state two conclusions.

1. The vision that partial evaluation ought to be able to produce the KMP from a naive string matcher is due to Yoshihiko Futamura, who used it as a guiding light to develop Generalized Partial Computation. We observe that this vision also encompasses other linear string matchers.
2. Neil Jones envisioned that polyvariant specialization ought to be enough to implement a self-applicable partial evaluator. We observe that this vision also applies to the derivation of linear string matchers by partial evaluation.

Acknowledgments: Each of the authors have benefited in one way or another from the research environment established by Neil Jones at DIKU and from his scientific approach to problem solving. Thanks are also due to Yoshihiko Futamura for proposing the initial problem, in 1987; to David Wise for his interest and encouragements, in 1989; and to Andrzej Filinski, Bernd Grobauer, and Julia Lawall for comments. This article has also benefited from Torben Mogensen’s proof-reading.

This work is supported by the NSF grant EIA-9806745 and by the ESPRIT Working Group APPSEM (www.md.chalmers.se/Cs/Research/Semantics/APPSEM/).

A Total correctness of the generic string matcher

In order to prove the correctness of the algorithm, to be stated as Theorem 1, we annotate the core program in Figure 1 with invariants.

A.1 Notation and properties

We use a `teletype font` to refer to the text of the core program and *italics* to refer to what this text denotes. So for example `p`, the first parameter of `match` in Figure 1, denotes a pattern p . We write \mathcal{I}_p for the set $\{0, \dots, lp - 1\}$, where lp denotes the length of p . Given a list of indices is , possibly with repetitions, we write \widehat{is} to denote the set of indices occurring in the list. Finally, for a base index bt , and noting that in each run, the pattern p and the text t are fixed, we write $FailsBefore(bt)$ if p is not a prefix of any suffix of t starting (strictly) before bt . (Therefore $FailsBefore(0)$ holds vacuously.)

We write $p[j]$ to denote the character in a pattern p at position j , starting from 0, and $t[bt+j]$ to denote the character in a text t at position $bt+j$. The two functions *cache-ref-pos?* and *cache-ref-neg?* test whether an entry in the cache is positive or negative, and the two functions *cache-ref-pos* and *cache-ref-neg* project a positive entry and a negative entry into the corresponding character and list of characters, respectively.

We introduce the following predicates to characterize the contents of the cache.

- The predicate $c \models^j x$ (read ‘ x agrees with c at j ’) tells us that the character x agrees with the cache c at index j :

$$c \models^j x \quad \text{iff} \quad \left\{ \begin{array}{l} \text{cache-ref-pos?}(c[j]) = \text{true} \Rightarrow \text{cache-ref-pos}(c[j]) = x \\ \text{cache-ref-neg?}(c[j]) = \text{true} \Rightarrow x \notin \widehat{js}, \text{ where} \\ \hspace{10em} js = \text{cache-ref-neg}(c[j]) \end{array} \right.$$

- The predicate $c \models p$ (read ‘ p agrees with c ’) generalizes the previous one in that it says that all the characters in a pattern p agree with the cache c :

$$c \models p \quad \text{iff} \quad \forall j \in \mathcal{I}_p . c \models^j p[j]$$

- The predicate $c \models p \setminus is$ (read ‘ p agrees with c but for is ’) says that except for the indices in is , all the characters in p agree with c :

$$c \models p \setminus is \quad \text{iff} \quad \forall j \in \mathcal{I}_p \setminus \widehat{is} . c \models^j p[j]$$

- The predicate $c \models_{bt} t$ (read ‘ t after bt agrees with c ’) says that the next lp characters in a text t after a base index bt agree with the cache c :

$$c \models_{bt} t \quad \text{iff} \quad \forall j \in \mathcal{I}_p . c \models^j t[bt + j]$$

Note that if c is empty (i.e., contains only empty lists of negative information), all the predicates above hold trivially. Moreover, if is is empty then $c \models p \setminus is$ implies $c \models p$. Finally, if $\widehat{is} = \mathcal{I}_p$ then $c \models p \setminus is$ holds vacuously.

We need the following assumptions about the auxiliary functions:

- *schedule-pc* returns a list of indices ranging over at least the non-empty entries in \mathcal{I}_p . That is, a list is such that $\forall j \in \mathcal{I}_p \setminus \widehat{is} : \text{cache-ref-neg?}(c[j]) = \text{true}$ with $\text{cache-ref-neg}(c[j]) = \text{nil}$.
- *schedule-pt* returns a list of indices including at least the negative entries in the cache (i.e., a list is such that $\widehat{is} \subseteq \mathcal{I}_p$ and satisfying $\forall j \in \mathcal{I}_p \setminus \widehat{is} . \text{cache-ref-pos?}(c[j]) = \text{true}$).
- If $c' = \text{cache-prune}(c)$ it must hold, for all $j \in \mathcal{I}_p$ and all x in the alphabet, that $c \models^j x$ implies $c' \models^j x$, as will be the case if, e.g., $c' = c$ or if c' is the empty cache.

A.2 Invariants

Invariant 0: By definition, the pattern and the cache have the same length, i.e.,

$$lp = \text{length}(p) = \text{length}(c).$$

Invariant 1: At the entry of *match-pc*, *loop-pc*, *match-pt*, and *loop-pt*, the length of the pattern (lp) and of the rest of the text to match (lt) satisfy

$$0 < lp \leq lt = \text{length}(t) - bt \leq \text{length}(t).$$

Also, whenever *loop-pt* and *loop-pc* are invoked, it is with indices ranging in the pattern and in the cache, i.e., $\widehat{is} \subseteq \mathcal{I}_p$. It follows from this invariant that for all $j \in \mathcal{I}_p$, $t[bt + j]$ is well-defined.

Invariant 2: At the entry of `match-pc`,

$$\left\{ \begin{array}{l} \text{FailsBefore}(bt + 1) \\ c \models_{bt+1} t \\ \text{cache-ref-neg?}(c[lp - 1]) = \text{true} \\ \text{cache-ref-neg}(c[lp - 1]) = \text{nil} \end{array} \right.$$

In words: p is not a prefix of any suffix of t before $bt + 1$, t after $bt + 1$ agrees with c , and the right-most entry of the cache is negative and empty.

Invariant 3: At the entry of `loop-pc`,

$$\left\{ \begin{array}{l} \text{FailsBefore}(bt + z) \\ c \models p \setminus is \\ c \models_{bt+z} t \\ 0 < z \leq lp \\ \text{cache-ref-neg?}(c[lp - z]) = \text{true}, \dots, \text{cache-ref-neg?}(c[lp - 1]) = \text{true} \\ \text{cache-ref-neg}(c[lp - z]) = \text{nil}, \dots, \text{cache-ref-neg}(c[lp - 1]) = \text{nil} \end{array} \right.$$

In words: p is not a prefix of any suffix of t before $bt + z$, p agrees with c but for is , t after $bt + z$ agrees with c , $lp - z$ is a valid index in p and c , and the z right-most entries of the cache are negative and empty.

Invariant 4: At the entry of `match-pt` and `loop-pt`,

$$\left\{ \begin{array}{l} \text{FailsBefore}(bt) \\ c \models p \\ c \models_{bt} t \end{array} \right.$$

In words: p is not a prefix of any suffix of t before bt , p agrees with c , and t after bt agrees with c .

Invariant 5: At the entry of `loop-pt`,

$$\forall j \in \mathcal{I}_p \setminus \widehat{is}. p[j] = t[bt + j]$$

In words: except for the indices in is , p equals t after bt . In particular, if is is empty, there is a match starting at bt .

A.3 Total correctness

Theorem 1 *The program in Figure 1, applied to a given p and a given t , terminates and returns an integer bt . If $bt \geq 0$, then*

- bt is a match, that is for all $j \in \mathcal{I}_p : p[j] = t[bt + j]$, and
- bt is the first match, that is $\text{FailsBefore}(bt)$ holds.

If $bt < 0$, then p is not a substring of t .

This result will clearly follow if we can prove the following properties:

Initialization: when `match-pt` is first called (from the main function `match`), the invariants hold;

Return conditions: when a function returns (without invoking another function), the invariants imply that the return value has the desired property;

Preservation of invariants: when one function (apart from `match`) invokes another function, the invariants are preserved; and

Termination: the program eventually terminates.

Let us address each of these issues in turn. First, however, it is convenient to note that Invariant 0 is valid, which is trivial, and that Invariant 1 is also valid, which follows since:

- lp does not change;
- if $0 < lp$ does not hold then the program stops without calling `match-pt`;
- the program checks $lp \leq lt$ both initially and each time lt changes, and always stops if the test fails;
- the equation $lt + bt = \text{length}(t)$ clearly holds initially, and continues to hold since each time bt increases, lt decreases by the same amount; and
- $bt \geq 0$, since bt is initially 0 and never decreases (since it is trivial to verify, as done later, that $z > 0$).

From now on, we thus only need to focus on establishing Invariants 2–5. Let us first state the following observation:

If $c \models_{bt} t$ then with $c' = \text{cache-shift}(c)$ the predicate $c' \models_{bt+1} t$ holds. (1)

We must show that for all $j \in \mathcal{I}_p$, the predicate $c' \models^j t[bt + 1 + j]$ holds. If $j = lp - 1$ this predicate trivially holds (since $c'[lp - 1]$ is empty), and otherwise $c'[j] = c[j + 1]$ so the predicate amounts to $c \models^{j+1} t[bt + 1 + j]$ which follows from $c \models_{bt} t$.

A.4 Initialization

We must prove that when `match` calls `match-pt`, Invariant 4 is established, that is with $c = \text{cache-init}(lp)$ we have

$$\left\{ \begin{array}{l} \text{FailsBefore}(0) \\ c \models p \\ c \models_0 t \end{array} \right.$$

which trivially holds.

A.5 Return conditions

The program may terminate in four places:

The function `match` returns 0: This return happens because p is empty. Then $\forall j \in \mathcal{I}_p : p[j] = t[0 + j]$ vacuously holds. Therefore, 0 is indeed the first match.

The function `match` returns -1: This return happens because t is shorter than p . Then clearly there cannot be a match.

The function `loop-pt` returns bt : This return happens because is is empty. Invariant 5 thus reads $\forall j \in \mathcal{I}_p : p[j] = t[bt + j]$ so bt is a match; and since Invariant 4 tells us that $FailsBefore(bt)$ holds, we see that bt is indeed the first match.

The function `loop-pc` returns -1: This return happens because $lt - z < lp$, which by Invariants 0 and 1 implies that

$$length(t) = lt + bt = lt - z + bt + z < length(p) + bt + z$$

showing that there cannot be a match starting at $bt + z$ or later. Since Invariant 3 tells us that there cannot be a match starting strictly before $bt + z$, we infer that p does indeed not occur in t .

A.6 Preservation of invariants

There are seven places in the program where a function (apart from the main function `match`) invokes another function; for each of these places we must verify that the invariants associated with the entry of the caller imply the invariants associated with the entry of the callee. This is enough since the program is tail-recursive, i.e., iterative.

The function `match-pt` calls `loop-pt`: Our assumption is that Invariant 4 holds at the entry of `match-pt`, and that in particular,

$$c \models p \text{ and } c \models_{bt} t \text{ hold.} \quad (2)$$

Clearly Invariant 4 still holds at the entry of `loop-pt`; we are left with showing that Invariant 5 also holds there. With $is = schedule-pt(c)$, we must establish that the following predicate holds:

$$\text{for all } j \in \mathcal{I}_p \setminus \widehat{is} : p[j] = t[bt + j]$$

For such a j , our assumptions on $schedule-pt$ imply that $cache-ref-pos?(c[j]) = true$, and by (2) we infer that with $x = cache-ref-pos(c[j])$ we have $x = p[j]$ and $x = t[bt + j]$, implying the desired $p[j] = t[bt + j]$.

The function `loop-pt` calls `loop-pt`: This invocation takes place because

$$x = p[i] = t[bt + i]. \quad (3)$$

We can assume that Invariants 4 and 5 hold at the beginning of the current invocation, and we must show that these invariants are preserved. With $is' = cdr(is)$ and with $c'' = cache-prune(c')$, where $c' = cache-extend-pos(c, i, x)$, the non-trivial part is to establish that the following predicates hold:

$$c'' \models p \tag{4}$$

$$c'' \models_{bt} t \tag{5}$$

$$\forall j \in \mathcal{I}_p \setminus \widehat{is'} : p[j] = t[bt + j] \tag{6}$$

For (6), we note that if $j = i$, the claim follows from (3), and that if $j \neq i$, $j \in \mathcal{I}_p \setminus \widehat{is'}$ and the claim follows from Invariant 5.

To establish (4) and (5), we note that by definition of *cache-extend-pos*, the predicate $c' \models^i x$ is satisfied and therefore $c' \models^i p[i]$ and $c' \models^i t[bt + i]$ are also satisfied. For $j \in \mathcal{I}_p \setminus \{i\}$, we deduce from Invariant 4 that the predicates $c' \models^j p[j]$ and $c' \models^j t[bt + j]$ are satisfied. Therefore the predicates $c' \models p$ and $c' \models_{bt} t$ are satisfied and the claim now follows from our assumptions on *cache-prune*.

The function `loop-pt` calls `match-pc`: This invocation takes place because

$$x = p[i] \neq t[bt + i]. \tag{7}$$

We can assume that Invariants 4 and 5 hold at the beginning of the current invocation, and in particular that the following predicates hold:

$$FailsBefore(bt) \tag{8}$$

$$c \models_{bt} t \tag{9}$$

With $c'' = cache-shift(c')$ where $c' = cache-extend-neg(c, i, x)$, our task is to show that the following predicates hold:

$$FailsBefore(bt + 1), \tag{10}$$

$$c'' \models_{bt+1} t, \tag{11}$$

$$cache-ref-neg?(c''[lp - 1]) = true, \text{ and}$$

$$cache-ref-neg(c''[lp - 1]) = nil.$$

The last two lines hold trivially, by definition of *cache-shift*. From (7) we see that

- there is no match starting at bt , which together with (8) implies (10); and that
- (9) can be strengthened to $c' \models_{bt} t$, which using (1) implies (11).

The function `match-pc` calls `loop-pc`: We can assume that Invariant 2 holds, and we must show that Invariant 3 holds with $z = 1$ and $is = schedule-pc(c)$. This is trivial; in particular $c \models p \setminus is$ holds since if $j \in \mathcal{I}_p \setminus \widehat{is}$ then by our assumptions on *schedule-pc*, $cache-ref-neg?(c[j]) = true$ with $cache-ref-neg(c[j]) = nil$, and therefore $c \models^j p[j]$.

The function `loop-pc` calls `match-pt`: We can assume that Invariant 3 holds at the entry of `loop-pc`. Therefore, and in particular, $FailsBefore(bt+z)$ and $c \models_{bt+z} t$ hold and $c \models p \setminus is$ also holds, which is equivalent to $c \models p$ since is is empty. But this is just what is needed to show that Invariant 4 holds at the entry of `match-pt`.

The function `loop-pc` invokes `(loop-pc z c (cdr is))`: The function `loop-pc` is called because either

- $cache-ref-pos?(c[i]) = true$ and $p[i] = cache-ref-pos(c[i])$, or
- $cache-ref-neg?(c[i]) = true$ and $p[i] \notin cache-ref-neg(c[i])$,

i.e., because

$$c \models^i p[i]. \quad (12)$$

We can assume that Invariant 3 holds at the entry of the current invocation, and we must show that it holds also at the entry of the new invocation. The only non-trivial case is to show that with $is' = cdr(is)$ we have $c \models p \setminus is'$, i.e., for all $j \in \mathcal{I}_p \setminus \widehat{is'}$ we must establish that $c \models^j p[j]$ is satisfied. But if $j = i$ this follows from (12), and if $j \neq i$ then $j \in \mathcal{I}_p \setminus \widehat{is}$ and the claim follows from Invariant 3.

The function `loop-pc` invokes `(loop-pc (+ z 1) c (schedule-pc c))`: The function `loop-pc` is called because (cf. the above case)

$$c \models^i p[i] \text{ does not hold.} \quad (13)$$

We can assume that Invariant 3 holds at the entry of the current invocation, and in particular that

$$FailsBefore(bt+z) \quad (14)$$

$$c \models_{bt+z} t \quad (15)$$

$$\begin{aligned} & cache-ref-neg?(c[lp-z]) = true, \\ & \dots, \text{ and} \end{aligned} \quad (16)$$

$$\begin{aligned} & cache-ref-neg?(c[lp-1]) = true \\ & cache-ref-neg(c[lp-z]) = nil, \\ & \dots, \text{ and} \end{aligned} \quad (17)$$

$$cache-ref-neg(c[lp-1]) = nil$$

Let $c' = cache-shift(c)$ and $is' = schedule-pc(c')$. From (13) and (17) we deduce that $i < lp - z$, implying $z < lp$ and hence $0 < z + 1 \leq lp$. From (16) and (17) we deduce, using the definition of $cache-shift$, that

$$\begin{aligned}
& \text{cache-ref-neg?}(c'[lp - z - 1]) = \text{true}, \\
& \dots, \\
& \text{cache-ref-neg?}(c'[lp - 1]) = \text{true}, \\
& \text{cache-ref-neg}(c'[lp - z - 1]) = \text{nil}, \\
& \dots, \text{ and} \\
& \text{cache-ref-neg}(c'[lp - 1]) = \text{nil}
\end{aligned}$$

From (15) and (1) we infer $c' \models_{bt+z+1} t$. Our assumptions about *schedule-pc* tell us that if $j \in \mathcal{I}_p \setminus \widehat{is'}$ then we have $\text{cache-ref-neg?}(c'[j]) = \text{true}$ with $\text{cache-ref-neg}(c'[j]) = \text{nil}$ and therefore $c' \models^j p[j]$, demonstrating that $c' \models p \setminus is'$ holds. By (15) we see that $c \models^i t[bt + z + i]$ holds, which combined with (13) tells us that $t[bt + z + i] \neq p[i]$ so there is no match starting at $bt + z$. From (14) we therefore deduce that *FailsBefore*($bt + z + 1$) holds. Collecting the above results, we see that Invariant 3 indeed holds at the entry of the new invocation of `loop-pc`.

A.7 Termination

Assume that all calls to `match-pt` and `match-pc` have been unfolded, leaving us with only `loop-pt` and `loop-pc`. We now assign a measure to each call as follows: to each call of `loop-pt` the measure is the triple $(lt, lp, \text{length}(is))$; to each call of `loop-pc` the measure is the triple $(lt, lp - z, \text{length}(is))$. Triples are ordered lexicographically, yielding (as the invariants show) a well-founded ordering with respect to which each iteration is strictly decreasing (as is easy to see). Termination follows, concluding the proof of Theorem 1.

References

- [1] Alfred V. Aho. Algorithms for finding patterns in strings. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, chapter 5, pages 255–300. The MIT Press, 1990.
- [2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
- [3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.
- [4] Maria Alpuente, Moreno Falaschi, Pascual Julià, and German Vidal. Specialization of inductively sequential functional logic programs. In Charles Consel, editor, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 151–162, Amsterdam, The Netherlands, June 1997. ACM Press.
- [5] Torben Amtoft. *Sharing of Computations*. PhD thesis, DAIMI, Department of Computer Science, University of Aarhus, 1993. Technical report PB-453.

- [6] Ricardo A. Baeza-Yates, Christian Choffrut, and Gaston H. Gonnet. On Boyer-Moore automata. *Algorithmica*, 12(4/5):268–292, 1994.
- [7] Guntis J. Barzdins and Mikhail A. Bulyonkov. Mixed computation and translation: Linearisation and decomposition of compilers. Preprint 791, Computing Centre of Siberian Division of USSR Academy of Sciences, Novosibirsk, Siberia, 1988.
- [8] Richard S. Bird. Improving programs by the introduction of recursion. *Communications of the ACM*, 20(11):856–863, November 1977.
- [9] Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors. *Partial Evaluation and Mixed Computation*. North-Holland, 1988.
- [10] Anselm Blumer, J. Blumer, David Haussler, Andrzej Ehrenfeucht, M. T. Chen, and Joel I. Seiferas. The smallest automaton recognizing the subwords of a text. *Theoretical Computer Science*, 40:31–55, 1985.
- [11] Anders Bondorf. Similix 5.0 manual. Technical report, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, 1993. Included in the Similix 5.0 distribution.
- [12] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.
- [13] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.
- [14] Robert S. Boyer and J. Strother Moore. *A Computational Logic*. ACM Monograph Series. Academic Press, 1979.
- [15] Mikhail A. Bulyonkov. Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21:473–484, 1984.
- [16] Sandrine Chirokoff, Charles Consel, and Renaud Marlet. Combining program and data specialization. *Higher-Order and Symbolic Computation*, 12(4):309–335, 1999.
- [17] Livio Colussi. Correctness and efficiency of pattern matching algorithms. *Information and Computation*, 95:225–251, 1991.
- [18] Charles Consel and Olivier Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30(2):79–86, January 1989.
- [19] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.

- [20] Charles Consel, Olivier Danvy, and Karoline Malmkjær. The abstraction and instantiation of string-matching programs. Unpublished manuscript, December 1989, and talks given at Stanford University, Indiana University, Kansas State University, Northeastern University, Harvard, Yale University, and INRIA Rocquencourt.
- [21] Charles Consel and François Noël. A general approach for run-time specialization and its application to C. In Guy L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 145–156, St. Petersburg Beach, Florida, January 1996. ACM Press.
- [22] Max Crochemore and Christophe Hancart. Pattern matching in strings. In Mikhail J. Atallah, editor, *Algorithms and Theory of Computation Handbook*, chapter 11. CRC Press, Boca Raton, 1998.
- [23] Olivier Danvy and Ulrik P. Schultz. Lambda-dropping: Transforming recursive equations into programs with block structure. *Theoretical Computer Science*, 248(1-2):243–287, 2000.
- [24] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
- [25] Andrei P. Ershov, Dines Bjørner, Yoshihiko Futamura, K. Furukawa, Anders Haraldsson, and William Scherlis, editors. *Special Issue: Selected Papers from the Workshop on Partial Evaluation and Mixed Computation, 1987*, New Generation Computing, Vol. 6, No. 2-3. Ohmsha Ltd. and Springer-Verlag, 1988.
- [26] Yoshihiko Futamura and Kenroku Nogi. Generalized partial computation. In Bjørner et al. [9], pages 133–151.
- [27] Robert Glück and Jesper Jørgensen. Generating optimizing specializers. In Henri Bal, editor, *Proceedings of the Fifth IEEE International Conference on Computer Languages*, pages 183–194, Toulouse, France, May 1994. IEEE Computer Society Press.
- [28] Robert Glück and Andrei Klimov. Occam’s razor in metacomputation: the notion of a perfect process tree. In Patrick Cousot, Moreno Falaschi, Gilberto Filé, and Antoine Rauzy, editors, *Proceedings of the Third International Workshop on Static Analysis WSA ’93*, number 724 in Lecture Notes in Computer Science, pages 112–123, Padova, Italy, September 1993. Springer-Verlag.
- [29] Robert Glück and Valentin F. Turchin. Application of metasystem transition to function inversion and transformation. In *Proceedings of the international symposium on symbolic and algebraic computation*, pages 286–287, Tokyo, Japan, August 1990. ACM, ACM Press.

- [30] Bernd Grobauer and Julia L. Lawall. Partial evaluation of pattern matching in strings, revisited. Technical report BRICS-RS-00-31, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, November 2000.
- [31] Manuel Hernández and David A. Rosenblueth. Development reuse and the logic program derivation of two string-matching algorithms. In Harald Søndergaard, editor, *Proceedings of the Third International Conference on Principles and Practice of Declarative Programming*, Firenze, Italy, September 2001. ACM Press. To appear.
- [32] Christoph M. Hoffman and Michael J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, 1982.
- [33] R. Nigel Horspool. Practical fast searching in strings. *Software—Practice and Experience*, 10(6):501–506, 1980.
- [34] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, 1993. Available online at <http://www.dina.kvl.dk/~sestoft/pebook/pebook.html>.
- [35] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In Jean-Pierre Jouannaud, editor, *Rewriting Techniques and Applications*, number 202 in Lecture Notes in Computer Science, pages 124–140, Dijon, France, May 1985. Springer-Verlag.
- [36] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. MIX: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.
- [37] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised⁵ report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998. Available online at <http://www.brics.dk/~hosoc/11-1/>.
- [38] Todd B. Knoblock and Erik Ruf. Data specialization. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 31, No 5, pages 215–225. ACM Press, June 1996.
- [39] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.
- [40] Laura Lafave and John P. Gallagher. Constraint-based partial evaluation of rewriting-based functional logic programs. In Norbert E. Fuchs, editor, *7th*

International Workshop on Program Synthesis and Transformation, number 1463 in Lecture Notes in Computer Science, pages 168–188, Leuven, Belgium, July 1997. Springer-Verlag.

- [41] Karoline Malmkjær. Program and data specialization: Principles, applications, and self-application. Master’s thesis, DIKU, Computer Science Department, University of Copenhagen, August 1989.
- [42] Karoline Malmkjær. *Abstract Interpretation of Partial-Evaluation Algorithms*. PhD thesis, Department of Computing and Information Sciences, Kansas State University, Manhattan, Kansas, March 1993.
- [43] Karoline Malmkjær and Olivier Danvy. Preprocessing by program specialization. In Uffe H. Engberg, Kim G. Larsen, and Peter D. Mosses, editors, *Proceedings of the 6th Nordic Workshop on Programming Theory*, pages 266–268, Department of Computer Science, University of Aarhus, October 1994. BRICS NS-94-4.
- [44] Jonathan Martin and Michael Leuschel. Sonic partial deduction. In Dines Bjørner, Manfred Broy, and Alexander V. Zamulin, editors, *Perspectives of System Informatics, Third International Andrei Ershov Memorial Conference*, number 1755 in Lecture Notes in Computer Science, pages 101–112, Akademgorodok, Novosibirsk, Russia, July 1999. Springer-Verlag.
- [45] Helmuth Partsch and Frank A. Stomp. A fast pattern matching algorithm derived by transformational and assertional reasoning. *Formal Aspects of Computing*, 2(2):109–122, 1990.
- [46] Christian Queinnec and Jean-Marie Geffroy. Partial evaluation applied to pattern matching with intelligent backtrack. In *Proceedings of the Second International Workshop on Static Analysis WSA ’92*, volume 81-82 of *Bigre Journal*, pages 109–117, Bordeaux, France, September 1992. IRISA, Rennes, France.
- [47] Robert Schaback. On the expected sublinearity of the Boyer-Moore algorithm. *SIAM Journal on Computing*, 17(4):648–658, 1988.
- [48] Donald A. Smith. Partial evaluation of pattern matching in constraint logic programming languages. In Paul Hudak and Neil D. Jones, editors, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, SIGPLAN Notices, Vol. 26, No 9, pages 62–71, New Haven, Connecticut, June 1991. ACM Press.
- [49] Morten Heine Sørensen. Turchin’s supercompiler revisited. an operational theory of positive information propagation. Master’s thesis, DIKU, Computer Science Department, University of Copenhagen, April 1994. DIKU Rapport 94/17.

- [50] Morten Heine Sørensen, Robert Glück, and Neil Jones. Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In Donald Sannella, editor, *Proceedings of the Fifth European Symposium on Programming*, number 788 in Lecture Notes in Computer Science, pages 485–500, Edinburgh, Scotland, April 1994. Springer-Verlag.
- [51] Morten Heine Sørensen, Robert Glück, and Neil D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.
- [52] Daniel M. Sunday. A very fast substring search algorithm. *Communications of the ACM*, 33(8):132–142, August 1990.
- [53] Masato Takeichi and Yoji Akama. Deriving a functional Knuth-Morris-Pratt algorithm. *Journal of Information Processing*, 13(4):522–528, 1990.
- [54] Bruce W. Watson. *Taxonomies and Toolkits of Regular Language Algorithms*. PhD thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands, 1995.
- [55] Peter Weiner. Linear pattern matching algorithms. In *IEEE Symposium on Switching and Automata Theory*, pages 1–11, New York, 1973.